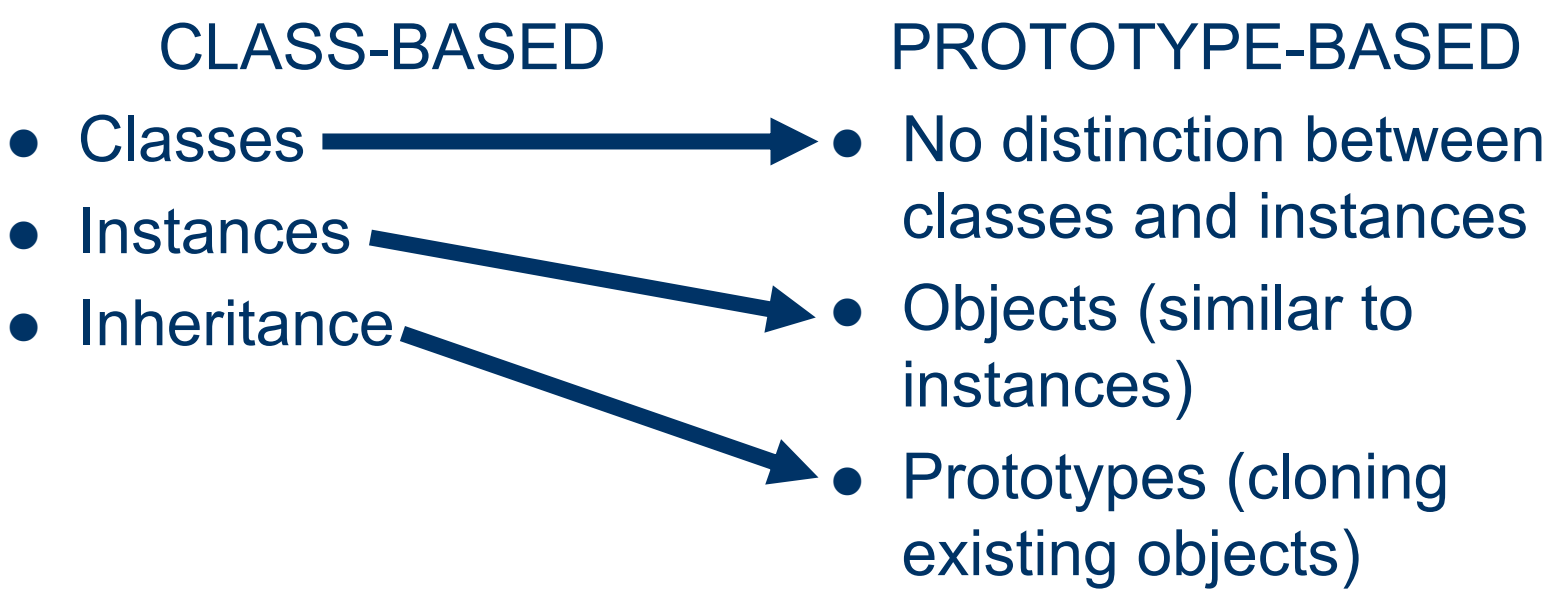
INHERITANCE, SUBTYPING, PROTOTYPES in Object-Oriented Languages

Orlin Grigorov
McMaster University
CAS 706, Fall 2006
ogrigorov@gmail.com

Two kinds of OO programming languages



Outline

- Inheritance
  - Single
  - Multiple
  - Interfaces
  - Mixins
- Subtyping
- Prototypes

Inheritance

Inheritance (definition)

- Refers to the fact that the definition of a new class can assume (or rely upon the existence of) another definition. Alternatively, it makes a previously defined structure available for incorporation in a new one.
  - A method of code sharing;
- If one class inherits from another, the inheriting class specializes a class which is more general.
  - Logical relationship between classes (specialization/generalization);
- Enables subtypes to be produced given a definition of a supertype.
  - Type-based account;

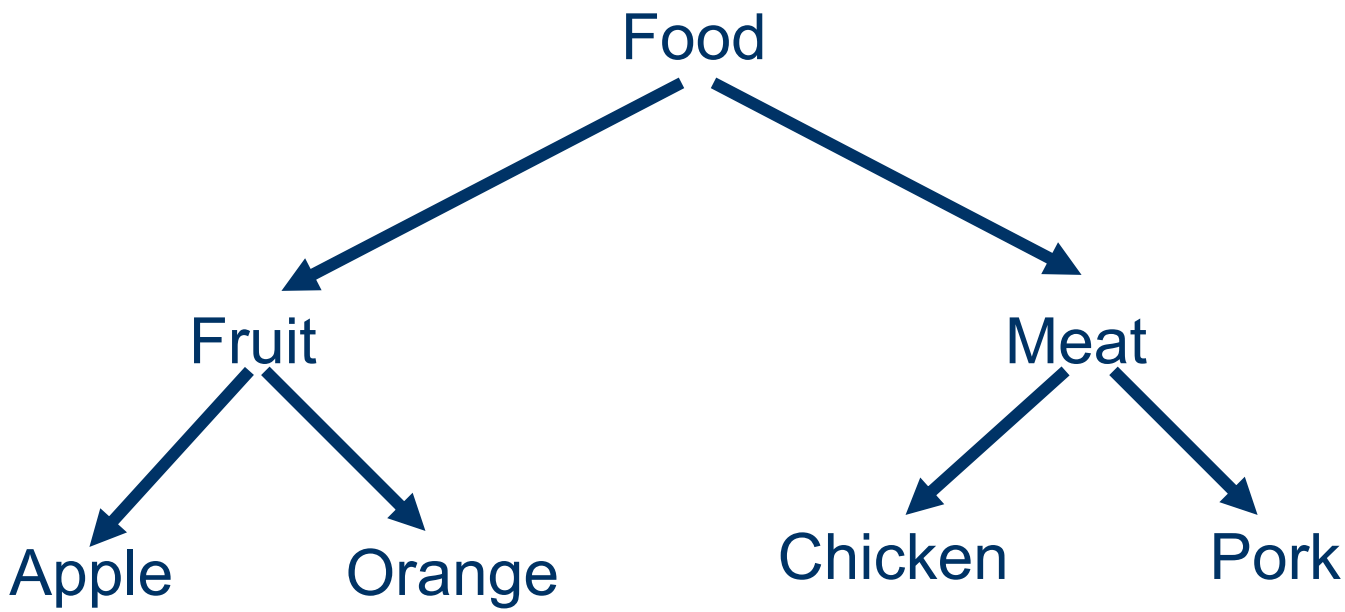
Inheritance (terminology)

- Subclass or child class or derived class: inherits all properties of its parent class, as well as adds other properties of its own. Think of it as:
  - An **extension** of the parent class (larger set of properties);
  - A **contraction** of the parent class (more specialized)

- Superclass or parent class or base class: a class that is one level higher in the class hierarchy than its child classes.

Inheritance is "*is a kind of*" relation

* The relation applies to classes, not instances!



Reasons to use inheritance

- Code reuse
  - Because a child class can inherit behaviour from a parent class, the code does not need to be rewritten for the child.
- Concept reuse
  - If a child overrides behaviour defined in the parent, although no code is shared, they share the definition of the method

Inheritance in various languages

- C++

```
class B : public A { .. }
```

- C#

```
class B : A { .. }
```

- CLOS

```
(defclass B (A) () )
```

- JAVA

```
class B extends A {
..
}
```

- Object Pascal

```
type
        B = object (A)
        ...
        end;
```

- Python

```
class B(A):
        def __init__(self):
        ...
```

- Ruby

```
class B < A
...
end
```

Inheritance in various languages (cont.):

- Java, Smalltalk, Objective-C, Delphi Pascal: TREES
  - Require every class to inherit from an existing parent class;
  - A single root that is ancestor to all objects (termed *Object* in Smalltalk and Objective-C, *TObject* in Delphi Pascal);
  - Any behaviour provided by the root → inherited in all objects;
  - Thus, every object is guaranteed to possess a common minimal level of functionality.

-continues-

Inheritance in various languages (cont.):

- **DISADVANTAGE:** combines all classes into a tightly coupled unit.

- C++ and Apple Pascal: FOREST
  - Do not require every class to inherit from a parent;
  - Not forced to carry a large library of classes, only a few of which may be used in any one program;
  - ..but no programmer-defined functionality that all objects are guaranteed to possess.

Tree vs. Forest

Tree

- Advantages:
  - Single class hierarchy;
  - Standardization: the functionality of the root is inherited by all objects—all have some basic functionality.
- Disadvantages:
  - Tight coupling;
  - Larger libraries.

Forest

- Advantages:
  - Many smaller hierarchies;
  - Smaller libraries of classes for application, less coupling possible.
- Disadvantages:
  - No shared functionality among all objects.

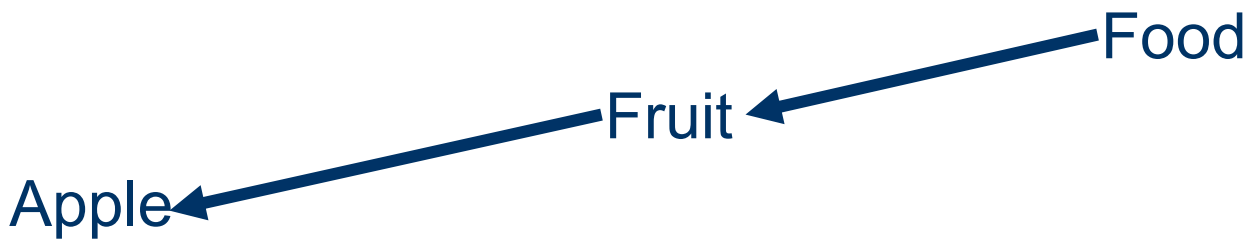
Single Inheritance

- A class can have only a single superclass;
- The data and behavior of a particular superclass is available to the subclass;

- **Advantages:**
  - Simple to include in the design
  - Code re-use is very simple to obtain

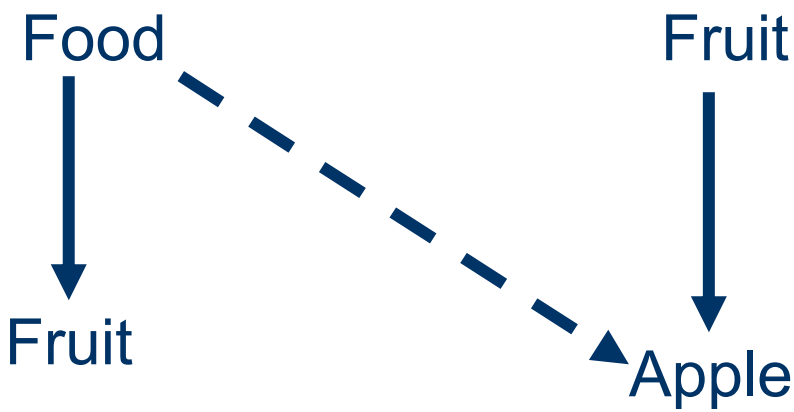
Single Inheritance (cont.)

- There is only 1 root class
- There is only a linear sequence of classes between *Apple* and *Food*



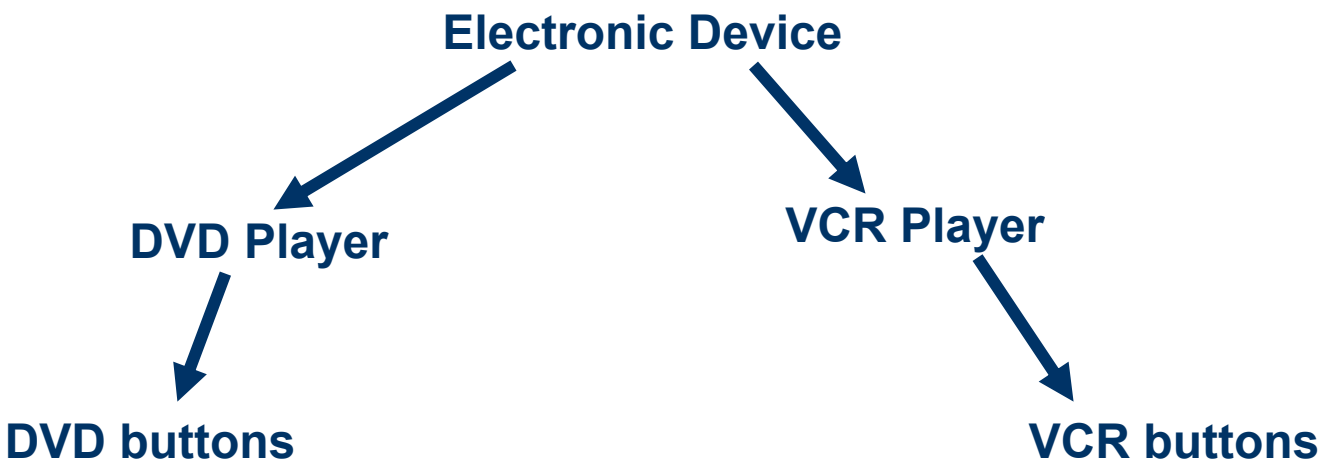
Single Inheritance (cont.)

- Property: it is always transitive



Single Inheritance (cont.)

- **Disadvantages:**
  - Some models cannot be accurately modeled using Single Inheritance;
  - Leads to duplicate code;



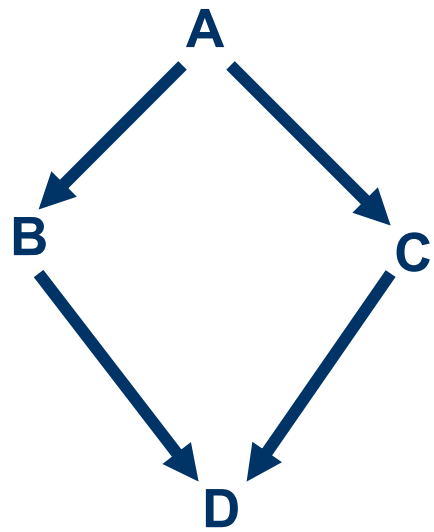
Multiple Inheritance

- A class can inherit from anywhere in a class hierarchy;
- A child class can have more than one parent;
- Provides more realistic framework;
  - e.g. A child has two biological parents.

- Available in some OO languages, such as C++ and Eiffel

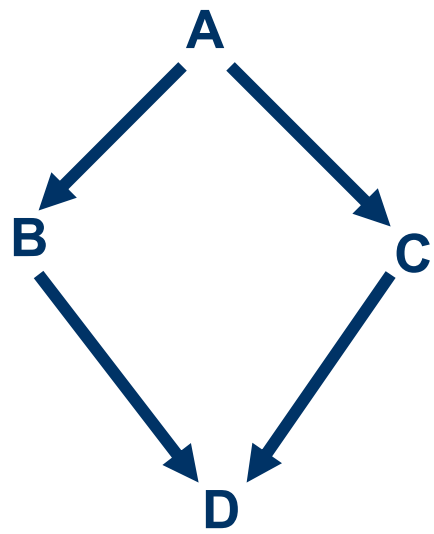
Multiple Inheritance (cont.)

- Structure is a **lattice** or **Directed Acyclic Graph**
- Problems:
  - Replication of inherited slots;
  - *Meaning* of the program of the program can be different depending on search algorithm.



…example (demonstrates concept, will not compile)

```
class A {
        abstract void printMe();
}

class B extends A {
        void printMe() { --“B”-- }
}

class C extends A {
        void printMe() { --”C”-- }
}

class D extends B, C {}

A d = new D();
d.printMe();  //problem, which one?!
```



C++, resolve manually: option 1

- Explicitly qualify the call by prefixing it with the name of the intended superclass using the scope resolution operator:

```
d.B::printMe();
```

OR

```
d.C::printMe();
```

C++, resolve manually: option 2

- Add a new method ***foo*** to class D:

```
class D : public B, public C {
     void foo(void) {
          B::printMe();  //one, the other,
          C::printMe();  //both or neither
     }
}
```

- Similar solution in Eiffel: use feature renaming, see next slide 

Eiffel

```
class D inherit
      B
      rename
            printMe as printMe_B
      select
            printMe_B
      end;
      C
      rename
            printMe as printMe_C
      end
      ...
end
```

This makes *printMe_B* as the version of *printMe* inherited from B. It also becomes the one that will be used in D.

C++: virtual

- Follows each inheritance path separately.
- …so a D object would actually contain two separate A objects;
- If the inheritance from A to B and from A to C are both marked “virtual”, creates only one A object.

```
class B : virtual A
```

- Uses of A's members work correctly.

Perl

- Perl handles this by specifying the inheritance classes as an ordered list.
- Class B and its ancestors would be checked before class C and its ancestors, so the method in A would be inherited through B.

Python

- Python had to deal with this upon the introduction of new-style classes, all of which have a common ancestor, object.
- Python creates a list of the classes that would be searched in left-first depth-first order (D, B, A, C, A) and then removes all but the last occurrence of any repeated classes.
- Thus, the method resolution order is:

  D, B, C, A.

Interfaces: Java

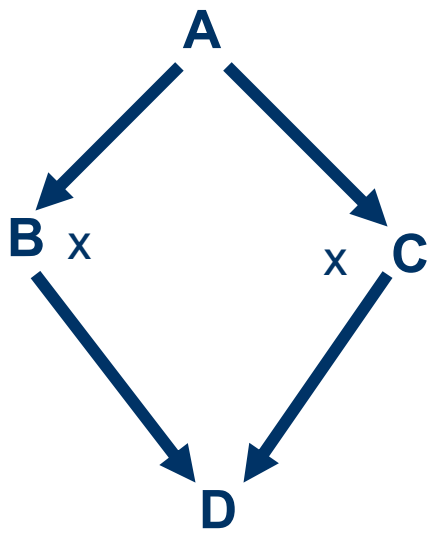
- Allows only single inheritance;
- Allows the multiple inheritance of interfaces.
- Interface ≠ Abstract class
  - In interfaces you cannot attach any behaviour, but can implement multiple interfaces;
  - …whereas in abstract classes you can define behaviour, but inherit only one abstract class.
- Also in Objective-C (protocols), PHP, C#.

Interfaces: example in Java

```
interface B {
        void printMe();
}

interface C {
        void printMe();
}

class A { ... }

class D extends A implements B {
        void printMe() { --”B”-- }
}
```

Alternative solution: Linearization

1. Flatten the inheritance structure into a linear chain without duplicates;
2. Search the chain in order to find slots (like in single inheritance);



→ B and C both define x ≡ printMe();

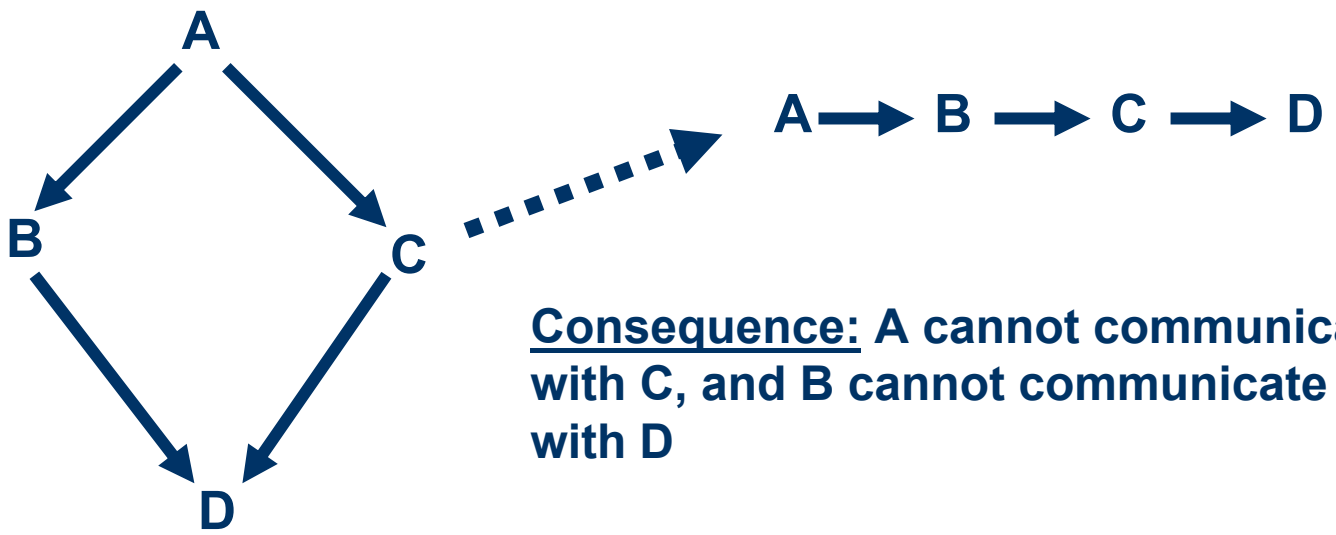
→One will be masked off by the algorithm.

* The selection is arbitrary unless more information is given.

- The order, in which the algorithm encounters superclasses is important;
- Programmer can set the order though.

Linearization: problem

- Inherited classes are not guaranteed communication with their direct ancestors;
- The algorithm can insert unrelated classes between an inheriting class and one of its direct ancestors:



<u>Consequence:</u> A cannot communicate directly with C, and B cannot communicate directly with D

Linearization: implementation

- Visit each node and put it into a list;
- Once the whole graph is traversed and all nodes collected—remove all duplicates.
- 2 ways to remove them:
  - Starting from the front
  - ...or the back.

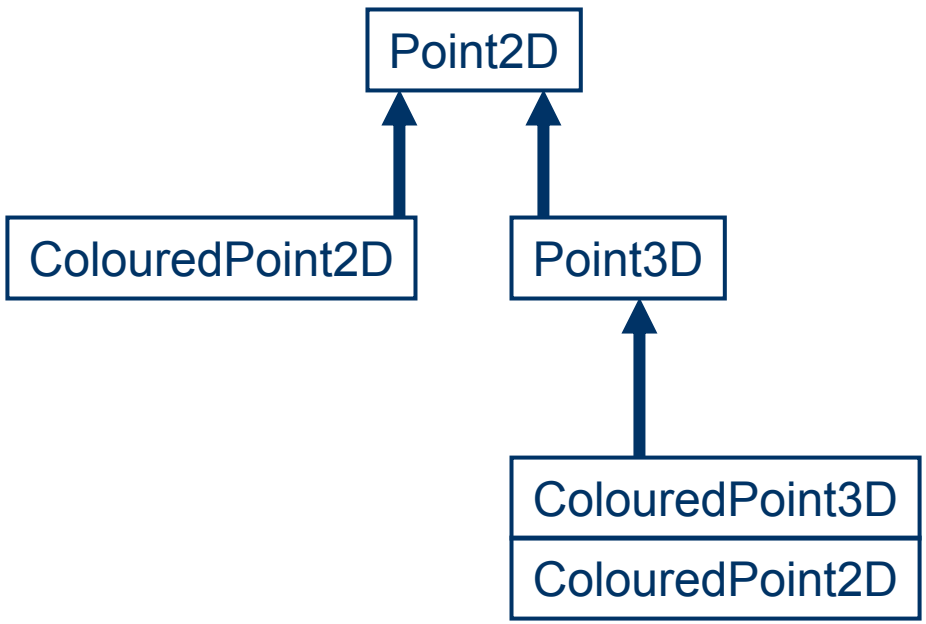
Mixins

- Referred to as “abstract subclasses”;
- They represent specification that may be applied to various parent classes to extend them with the same set of features.
- Available in Lisp, Scala, Ruby, Smallscript and CLOS.
  - Has been ported to Java as well.
- Alternative approach to multiple inheritance.

Mixins: example in Scala

```
class Point2D(xc: Int, yc: Int) {
        val x = xc;
        val y = yc;  //plus methods for manipulating 2D points
}

class ColouredPoint2D(u: Int, v: Int, c: String)
   extends Point2D(u, v) {
        var colour = c;
        def setColour(newCol: String): Unit = colour = newCol;
}

class Point3D(xc: Int, yc: Int, zc: Int)
   extends Point2D(xc, yc) {
        val z = zc;  //plus methods for manipulating 3D points
}

class ColouredPoint3D(xc: Int, yc: Int, zc: Int, col: String)
   extends Point3D(xc, yc, zc)
   with ColouredPoint2D(xc, yc, col);
```

Mixins: example in Scala (cont.)



Subtyping

Subtyping

- Subtyping consists of the rules by which objects of one type (class) are determined to be acceptable in contexts that expect another type (class).
  - The rules determine the legality of programs
- Subtyping should be based on the behaviour of objects.

Subtyping

- RULES:
  1) If instances of class B meet the external interface of class A, then B should be a subtype of A.
  2) One can use an instance of class B whenever an instance of class A is needed.
- Guarantee: always the same behaviour!

Subtyping vs. Interface Inheritance

- Subtyping
  - If B is a subtype of A, safe to use B instead of A → always the same behaviour.
- Interface Inheritance
  - If B has simply inherited the interface of A, then it is legal to use B instead of A, but no guarantee that it will do what you want.

Subtyping should not be equated with Inheritance

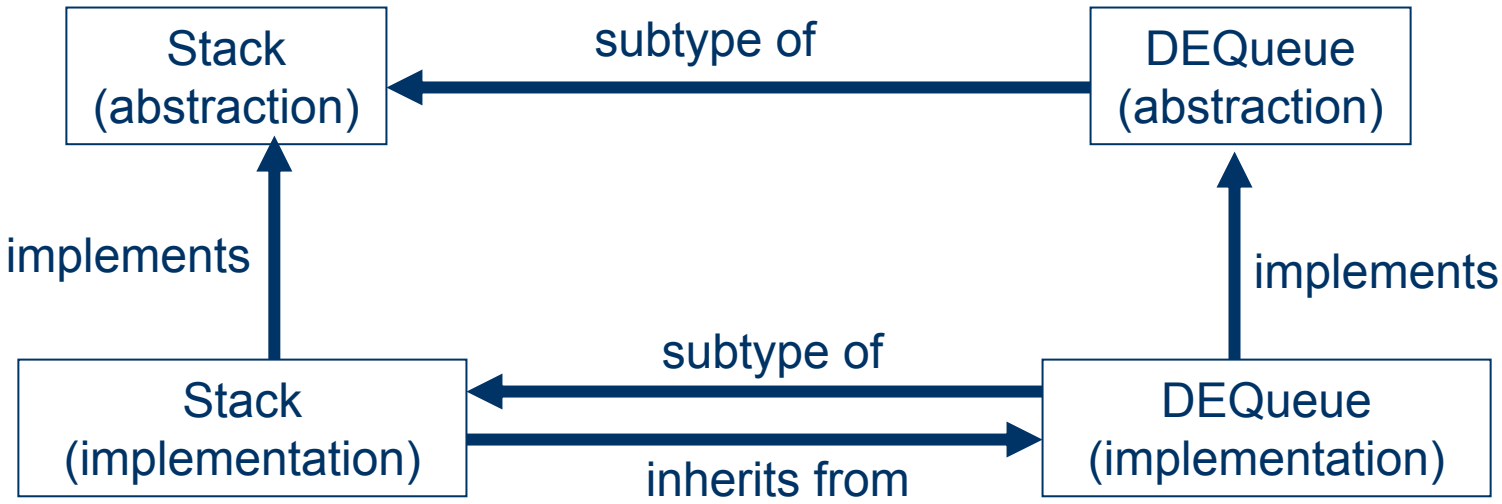
→ Consider *Stack* as a subtype of *DEQueue* (double-ended queue)

- In Trellis/Owl, Simula and Sather, class *Stack* is a subtype of *DEQueue* iff *Stack* is a subclass of *DEQueue.*
- If *Stack* is re-implemented, so that it becomes self-contained and inherits from no parent class, then the assumption that *Stack* is a subtype of *DEQueue* will no longer be legal.

➔Implementation hierarchy need not be the same as the type hierarchy.

Subtyping should not be equated with Inheritance (cont.)

- *Stack* inherits from *DEQueue*, but is not a subtype of it (excludes the operation that adds elements to the back of the queue).
- *DEQueue* is a subtype of *Stack*, but does not inherit from *Stack.*



Subtyping in various languages

- Statically typed languages: subtyping rules are extremely important as they determine the legality of programs.
  - e.g. C++, Java
- Dynamically typed languages: subtyping rules affect the result of type predicates.
  - e.g. CLOS, Smalltalk

Prototypes

Prototypes

- Object-Oriented programming without classes!
- No distinction between classes and instances.
- There are only objects, which are similar to instances.
- e.g.Javascript, Lua, Self, Kevo, NewtonScript, Mica, Obliq, Factor, Io, Lisaac, REBOL, Agora, Cecil and many others.
- “prototype-based programming” ≡ “object-based programming”

Prototypes (cont.)

- Create new objects by cloning, not instantiation.
- Analogy:
  - *Building an object in a class-based language is like building a house from a plan, while building an object in a prototype-based language is like building a house like the neighbors.*

Side by side: Java and JavaScript

Java

```
public class Employee {
      public String name;
      public String dept;
      public Employee () {
            this.name = "";
            this.dept = "general"; }
}
public class Manager extends Employee {
      public Employee[] reports;
      public Manager () {
            this.reports = new
                  Employee[0]; }
}
public class WorkerBee extends
      Employee {
      public String[] projects;
      public WorkerBee() {
            this.projects = new String[0];
      }
}
```

JavaScript

```
function Employee () {
      this.name = "";
      this.dept = "general";
}

function Manager () {
      this.reports = [];
}

Manager.prototype = new Employee;

function WorkerBee() {
      this.projects = [];
}

WorkerBee.prototype= new Employee;
```

Java vs. JavaScript

Class-based	Prototype-based
Class and instance are distinct entities.	All objects are instances.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the new operator.	The same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies all properties of all instances of a class. No way to add properties dynamically at runtime.	Constructor function or prototype specifies an initial set of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

References

- Budd, Timothy. An Introduction to Object-Oriented Programming. 3rd ed. Addison-Wesley, 2002.
- Craig, Iain. The Interpretation of Object-Oriented Programming Languages. Springer, 1999.
- "Multiple Inheritance." Wikipedia. 13 Nov. 2006 <http://en.wikipedia.org/wiki/Multiple_inheritance>.
- "Prototype-Based Programming." Wikipedia. 13 Nov. 2006 <http://en.wikipedia.org/wiki/Prototype-based_programming>.
- Merizzi, Nicholas. "Object Oriented, Classes, Objects, Inheritance, and Typing." Jan. 2005. 13 Nov. 2006.
- Sykes, Ed. "Object Oriented Languages." 13 Nov. 2006.

END

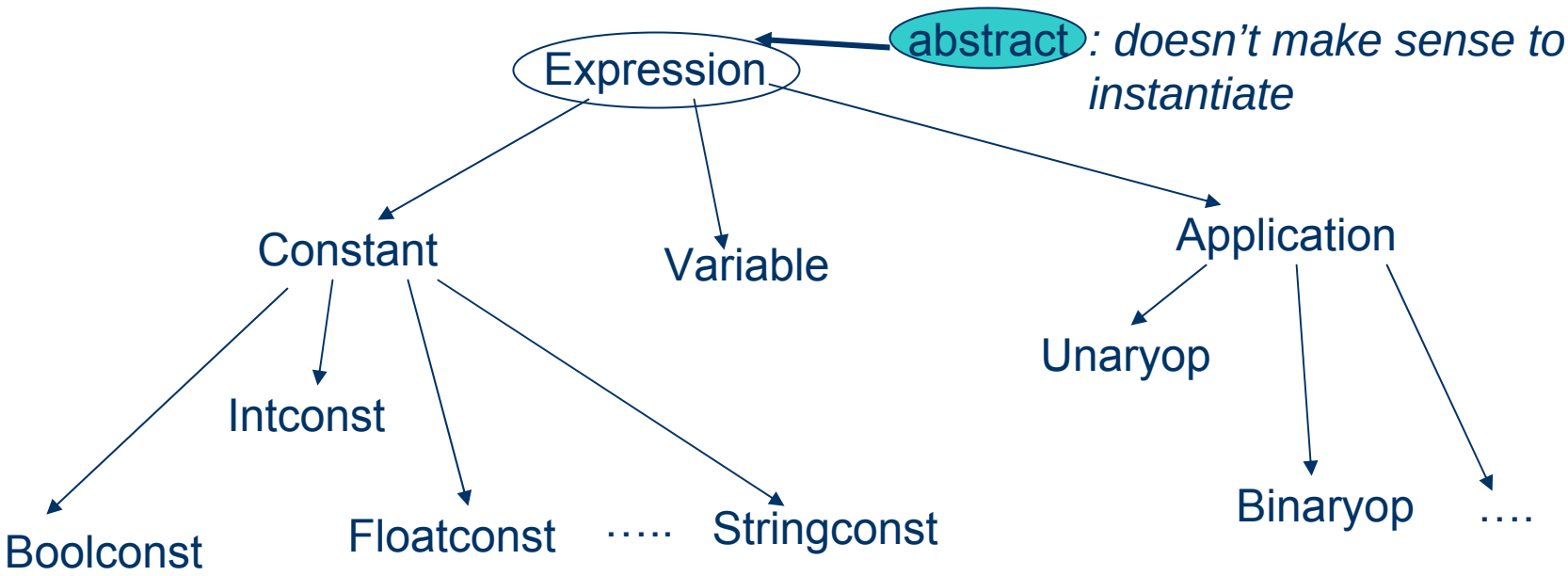
Time for questions and discussions…

Cut slides follow..

Abstract Classes

• Serve as place holders in an inheritance graph;
• Define only properties and behaviours required by many other classes;

- Cannot be instantiated!
- e.g. AST for expressions



Abstract Classes in various languages

- Some languages, like Java and Dylan, allow abstract classes (and methods) to be explicitly marked.
  - Java:

```
public abstract class A {
        public abstract void b(..);
        .
        .
}
```

  - Similar marking scheme in Dylan.

- In Eiffel: distinction achieved by means of the *deferred* annotation, which means that implementation of a feature is deferred until a later class.

Abstract Classes in various languages (cont.)

- In C++, a method can be defined as being virtual void.
  - The method is a virtual void, but..
  - its implementation is deferred to the subclasses of the class.

```
virtual int add1(int y) = 0
```

  - The “=0” is the “void” part of “virtual void”.
- No class annotation for an abstract class in C++.